Python for Machine Learning: Indexing and Slicing for Lists, Tuples, Strings, and other Sequential Types

Sergii Boiko

3 October, 2018

List is arguably the most useful and ubiquitous type in Python. One of the reasons it's so handy is Python slice notation. In short, slicing is a flexible tool to build new lists out of an existing list.

Python supports slice notation for any sequential data type like lists, strings, tuples, bytes, bytearrays, and ranges. Also, any new data structure can add its support as well. This is greatly used (and abused) in NumPy and Pandas libraries, which are so popular in Machine Learning and Data Science. It's a good example of "learn once, use everywhere".

In this article, we will focus on indexing and slicing operations over Python's lists. Most of the examples we will discuss can be used for any sequential data type. Only mutable assignment and deletion operations are not applicable to immutable sequence types like tuples, strings, bytes, and ranges.

Indexing

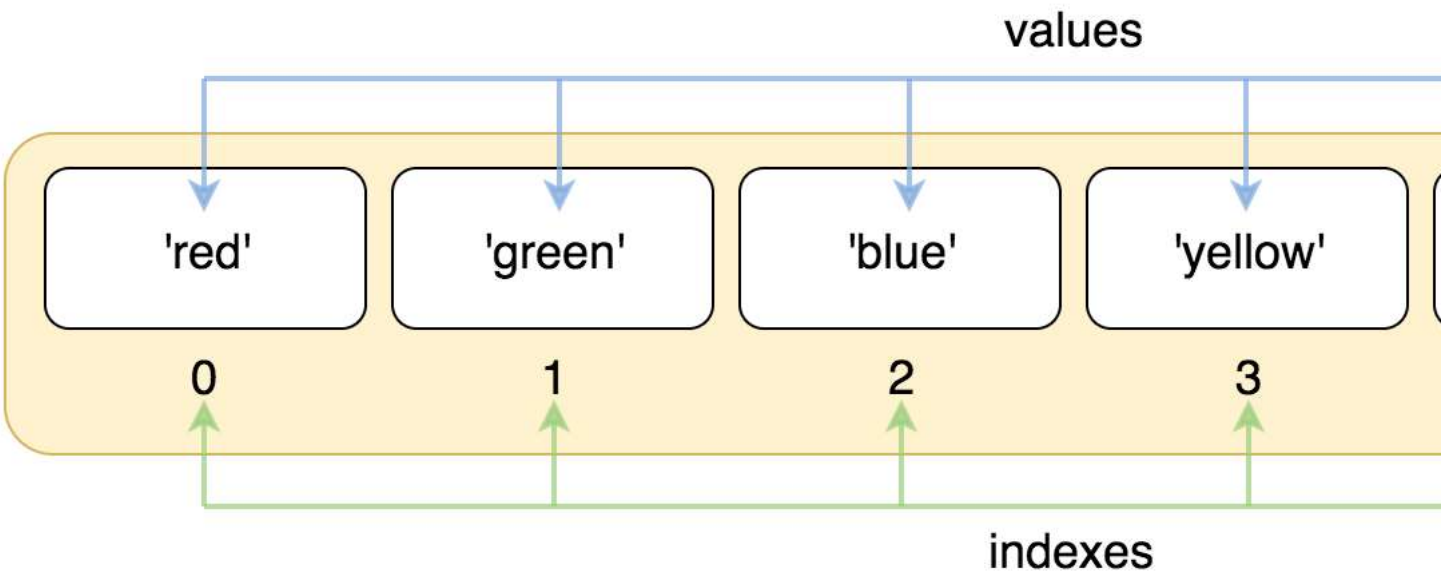
Before discussing slice notation, we need to have a good grasp of indexing for sequential types.

In Python, list is akin to arrays in other scripting languages(Ruby, JavaScript, PHP). It allows you to store an enumerated set of items in one place and access an item by its position – index.

Let's take a simple example:

```
1.   >>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
```

Here we defined a list of colors. Each item in the list has a value(color name) and an index(its position in the list). Python uses **zero-based** indexing. That means, the first element(value ‘red’) has an index 0, the second(value ‘green’) has index 1, and so on.



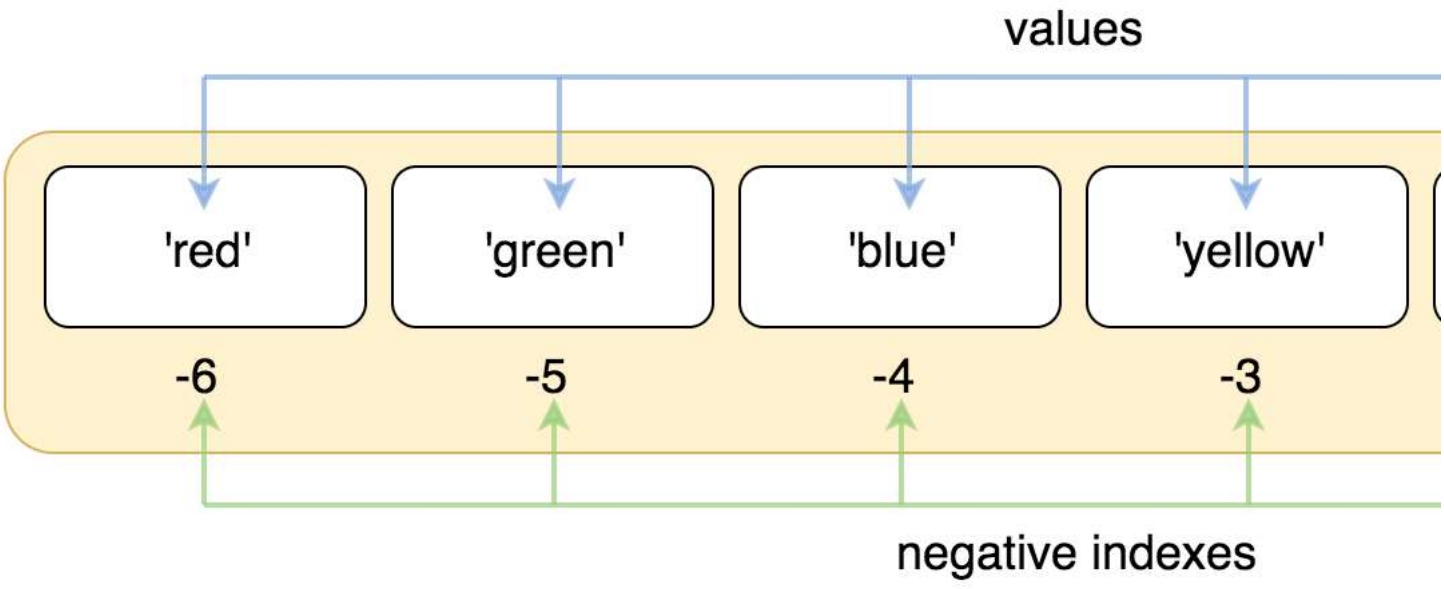
To access an element by its index we need to use square brackets:

```
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
>>> colors[0]
'red'
>>> colors[1]
'green'
>>> colors[5]
'black'
```

Negative indexes

Using indexing we can easily get any element by its position. This is handy if we use position from the head of a list. But what if we want to take the last element of a list? Or the penultimate element? In this case, we want to enumerate elements from the tail of a list.

To address this requirement there is negative indexing. So, instead of using indexes from zero and above, we can use indexes from -1 and below.



In negative indexing system -1 corresponds to the last element of the list(value ‘black’), -2 to the penultimate (value ‘white’), and so on.

```
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black']
>>> colors[-1]
'black'
>>> colors[-2]
'white'
>>> colors[-6]
'red'
```

Assignment

Before we used indexing only for accessing the content of a list cell. But it’s also possible to change cell content using an assignment operation:

```
>>> basket = ['bread', 'butter', 'milk']
```

```
>>> basket[0] = 'cake'
>>> basket
['cake', 'butter', 'milk']
>>> basket[-1] = 'water'
>>> basket
['cake', 'butter', 'water']
```

We can freely use positive or negative indexing for assignment.

Deletion

We can also easily delete any element from the list by using indexing and `del` statement:

```
>>> basket = ['bread', 'butter', 'milk']
>>> del basket[0]
>>> basket
['butter', 'milk']
>>> del basket[1]
>>> basket
['butter']
```

Indexing for other Sequential Types

Read-only indexing operations work perfectly well for all sequential types. But assignment and deletion operations are not applicable to immutable sequential types.

Slice Notation

As it was shown, indexing allows you to access/change/delete only a single cell of a list. What if we want to get a sublist of the list. Or we want to update a bunch of cells at once? Or we want to go on a frenzy and extend a list with an arbitrary number of new cells in any position?

Those and lots of other cool tricks can be done with slice notation. Let's look at this subject.

Basic Usage of Slices

Let's create a basic list:

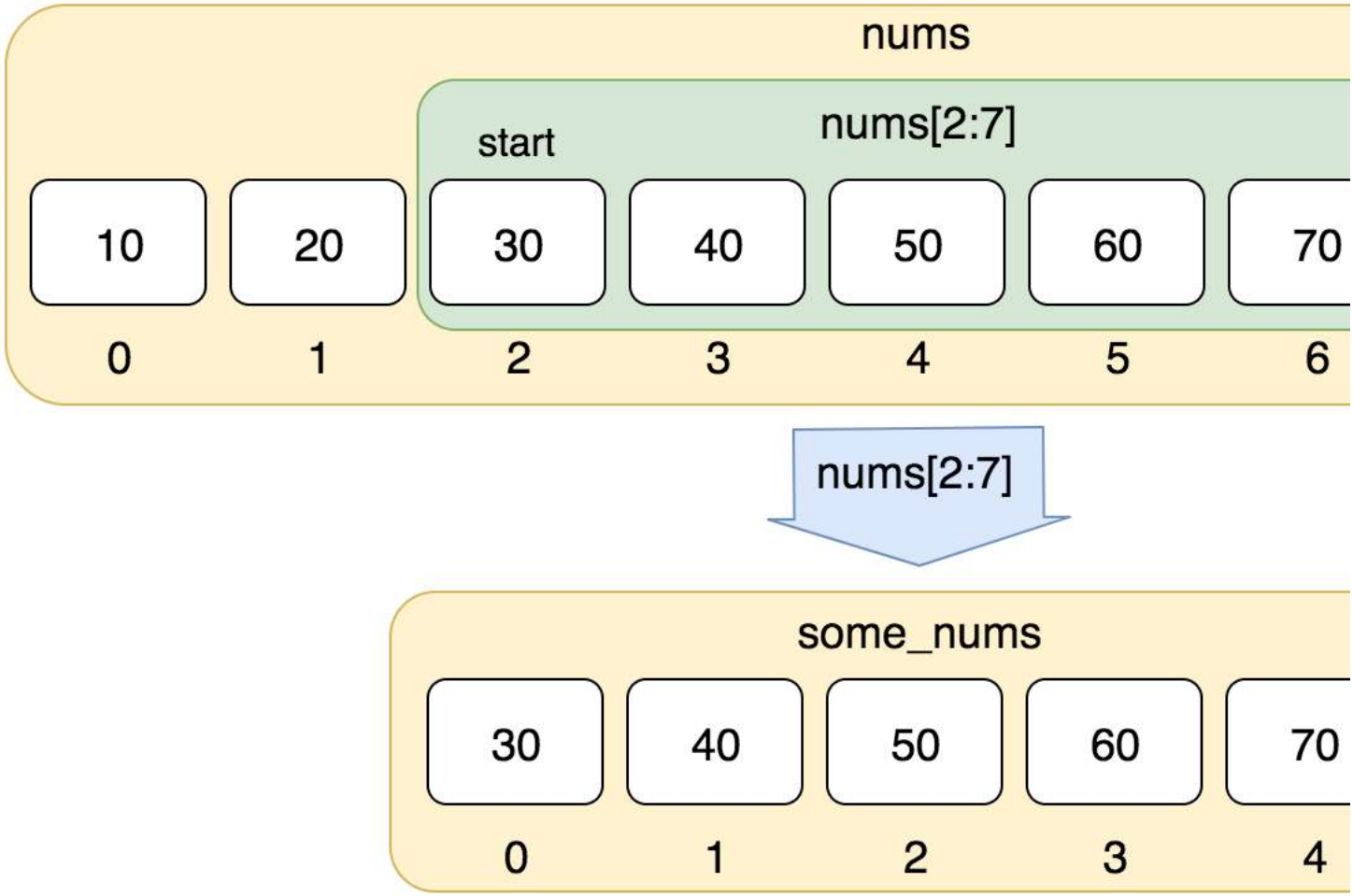
```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

What if we want to take a sublist from the `nums` list? This is a snap when using slice:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> some_nums = nums[2:7]
>>> some_nums
[30, 40, 50, 60, 70]
```

So, here is our first example of a slice: *2:7*. The full slice syntax is: *start:stop:step*. `start` refers to the index of the element which is used as a start of our slice. `stop` refers to the index of the element we should stop just before to finish our slice. `step` allows you to take each nth-element within a *start:stop* range.

In our example `start` equals `2`, so our slice starts from value `30`. `stop` is `7`, so the last element of the slice is `70` with index `6`. In the end, slice creates a new list(we named it `some_nums`) with selected elements.



We did not use `step` in our slice, so we didn't skip any element and obtained all values within the range.

With slices we can extract an arbitrary part of a list, e.g.:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[0:4]
[10, 20, 30, 40]
```

Here we start from the first element(index `0`) and take a list till the element with index `4`.

Taking n first elements of a list

Slice notation allows you to skip any element of the full syntax. If we skip the `start` number then it starts from `0` index:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[:5]
[10, 20, 30, 40, 50]
```

So, *nums[:5]* is equivalent to *nums[0:5]*. This combination is a handy shortcut to take n first elements of a list.

Taking n last elements of a list

Negative indexes allow us to easily take n-last elements of a list:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[-3:]
[70, 80, 90]
```

Here, the `stop` parameter is skipped. That means you take from the `start` position, till the end of the list. We start from the third element from the end (value `70` with index `-3`) and take everything to the end.

We can freely mix negative and positive indexes in `start` and `stop` positions:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[1:-1]
[20, 30, 40, 50, 60, 70, 80]
>>> nums[-3:8]
[70, 80]
>>> nums[-5:-1]
[50, 60, 70, 80]
```

Taking all but n last elements of a list

Another good usage of negative indexes:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[:-2]
[10, 20, 30, 40, 50, 60, 70]
```

We take all but the last two elements of original list.

Taking every nth-element of a list

What if we want to have only every 2-nd element of `nums`? This is where the `step` parameter comes into play:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[::2]
[10, 30, 50, 70, 90]
```

Here we omit `start`/`stop` parameters and use only `step`. By providing `start` we can skip some elements:

```
>>> nums[1::2]
[20, 40, 60, 80]
```

And if we don't want to include some elements at the end, we can also add the `stop` parameter:

```
>>> nums[1:-3:2]
[20, 40, 60]
```

Using Negative Step and Reversed List

We can use a negative `step` to obtain a reversed list:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[::-1]
[90, 80, 70, 60, 50, 40, 30, 20, 10]
```

Negative `step` changes a way, slice notation works. It makes the slice be built from the tail of the list. So, it goes from the last element to the first element. That's why we get a reversed list with a negative step.

Due to this peculiarity, `start` and `stop` should be provided from right to left as well. E.g., if you want to have a reversed list which starts from `80`:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[-2::-1]
[80, 70, 60, 50, 40, 30, 20, 10]
```

So, we start from the `-2` element (value `80`) and go from right to left collecting all the elements in a reversed list.

We can use `stop` value to stop taking before some element. E.g., let's not include `20` and `10` values:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[-2:1:-1]
[80, 70, 60, 50, 40, 30]
```

We use `1` for `stop` index, which is the element with value `20`. So, we go from 80 till 30, not including value `20`.

It's a bit baffling that with a negative `step`, `stop` index is located before `start`. Negative `step` turns everything upside down.

Of course, we can use an arbitrary negative `step`:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[-2:1:-3]
[80, 50]
```

Slice and Copying

One important thing to notice – is that list slice creates a shallow copy of the initial list. That means, we can safely modify the new list and it will not affect the initial list:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> first_five = nums[0:5]
>>> first_five[2] = 3
>>> first_five
[10, 20, 3, 40, 50]
>>> nums
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Despite our mutating the element under index `2`, it does not affect the `nums` list, because `first_five` list – is a partial copy of `nums` list.

There is the shortest form of slice notation – just colons *nums[:]*.

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums_copy = nums[:]
>>> nums_copy[0] = 33
>>> nums_copy
[33, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

It creates a shallow copy of the whole list and is a good shorthand when you need a copy of the original list.

Slice Object

But what if we want to use the same slice over and over again. Is there a way to create a slice object instead of using just the syntactic form?

This can be done using `slice` function:

```
>>> five_items_after_second = slice(2, 2 + 5)
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> colors = ['red', 'green', 'blue', 'yellow', 'white', 'black', 'silver']
>>> nums[five_items_after_second]
[30, 40, 50, 60, 70]
>>> colors[five_items_after_second]
['blue', 'yellow', 'white', 'black', 'silver']
```

`slice` function accepts arguments in the same order as in slice notation, and if you need to skip some element, just use `None`:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> all_but_two_last = slice(None, -2)
>>> nums[all_but_two_last]
[10, 20, 30, 40, 50, 60, 70]
>>> reversed = slice(None, None, -1)
>>> nums[reversed]
[90, 80, 70, 60, 50, 40, 30, 20, 10]
```

Slice Assignment

Python supports slice assignment operation, which allows us to make a bunch of neat operations over an existing list. Unlike previous slice operations, these mutate the original object in place. That's why they are not applicable to immutable sequential types.

Substitute part of a list

Slice assignment allows you to update a part of a list with new values:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[:4] = [1,2,3,4]
>>> nums
[1, 2, 3, 4, 50, 60, 70, 80, 90]
```

Here we do not change the number of elements in the list. Only some list values are updated.

Replace and Resize part of the list

We can replace part of a list with a bigger chunk instead:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[:4] = [1,2,3,4,5,6,7]
>>> nums
[1, 2, 3, 4, 5, 6, 7, 50, 60, 70, 80, 90]
```

In this case we extend the original list.

It's also possible to replace a bigger chunk with a smaller number of items:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[:4] = [1]
>>> nums
[1, 50, 60, 70, 80, 90]
```

Replace Every n-th Element

Adding `step` allows to replace each n-th element with a new value:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[::2] = [1,1,1,1,1]
>>> nums
[1, 20, 1, 40, 1, 60, 1, 80, 1]
```

Using slice assignment with `step` sets a limitation on the list we provide for assignment. The provided list should exactly match the number of elements to replace. If the length does not match, Python throws the exception:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[::2] = [1,1,1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 3 to extended slice of size 5
</module></stdin>
```

We can also use negative `step`:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[::-2] = [1,2,3,4,5]
>>> nums
[5, 20, 4, 40, 3, 60, 2, 80, 1]
```

By providing `start` and `stop` values we can narrow the replacement area:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> nums[1:5:2] = [2, 4]
>>> nums
[10, 2, 30, 4, 50, 60, 70, 80, 90]
```

Slice Deletion

We can also use `del` statement to remove a slice out of a list:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> del nums[3:7]
>>> nums
[10, 20, 30, 80, 90]
```

Here we've removed a bunch of elements in the middle of `nums` list.

We can also provide `step` parameter to slice and remove each n-th element:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> del nums[::2]
>>> nums
[20, 40, 60, 80]
```

With the full syntax, we can set boundaries for the elements to be removed:

```
>>> nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> del nums[1:7:2]
>>> nums
[10, 30, 50, 70, 80, 90]
```

So, we start deletion from 20(index 1) and remove each 2-nd element till the value 80(index 7).

And because slice deletion mutates the underlying object, it's not applicable to immutable sequential types.

Summary

We discussed two key list operations: indexing and slicing. Both concepts are crucial to efficient Python use.

This article prepared background for tackling indexing and slicing in `NumPy` `ndarrays` and `Pandas` `Series` and `DataFrame` objects. There are a lot of similarities between these structures, but there are also a lot of subtle differences and gotchas which will be discussed in upcoming articles dedicated to the machine learning techniques the Railsware team makes use of.

If you are still a beginner, here is the list of libraries and tools which may help you to start off with Machine Learning and AI.

- Mailtrap
- Jira Checklist
- Notemate

Contact

contact@railsware.com +1 (646) 397 4918 Contact Us
Krakow, Poland Szlak 50/A4, 31153
Kyiv, Ukraine Hryhoriia Skovorody St., 19,
BC “Podil Plaza”, 01004
Dubai, UAE Dubai Internet City Building 16, #48
Danbury, USA 118 Coalpit Hill Roads, 06810 CT

Railsware is a leading Ruby on Rails company which provides contracting and consulting services around the world. We are among the premium ruby on rails companies on the US market and our ROR development company is the exeperts at Ruby, Rails, HTML 5, and CSS3.